Programmation des machines à mémoire distribuée par distribution des données : langages et compilateurs

Françoise André, Olivier Chéron, Marc Le Fur, Yves Mahéo, Jean-Louis Pazat

SYNTHÈSE

Programmation des machines à mémoire distribuée par distribution de données : langages et compilateurs

Françoise André, Olivier Chéron, Marc Le Fur, Yves Mahéo et Jean-Louis Pazat

IRISA, 35042 Rennes Cedex

*RÉSUMÉ. La puissance et l'extensibilité des architectures parallèles à mémoire distribuée (*APMD*) satisfont a priori les besoins d'une large classe d'applications de calcul scientifique. Cependant, la programmation des* APMD *reste difficile. Dans cet article nous décrivons les recherches en cours visant à simplifier leur utilisation. Garder un modèle de programmation séquentiel et un espace de nommage global tout en spécifiant la distribution des données semble être une approche intéressante : l'utilisateur n'a pas à gérer de processus communicants qui sont produits automatiquement par un compilateur. Les principes généraux régissant la compilation sont la génération de code SPMD et l'application de la règle des écritures locales. Nous présentons trois projets de recherche (Vienna Fortran, Fortran D et Pandore) basés sur ces principes. Leurs langages sources et les optimisations développées sont décrits et comparés.*

ABSTRACT. The computing power and the scalability of distributed memory machines (DMM) fulfill the need of performance of a wide range of scientific applications; however, these machines are difficult to program. This paper presents approaches currently investigated to simplify the use of DMM. Keeping a sequential programming model operating on a global name space while specifying the distribution of data seems to be an interesting approach: the user has not to handle parallel communicating processes which are automatically generated by a compiler. The basic compiling technique relies on the production of SPMD code according to the owner compute rule. Three research projects based on this principle are described: Vienna Fortran, Fortran D and Pandore. Their source languages and the various developed optimizations are explained and compared.

MOTS-CLÉS : compilation, machines à mémoire distribuée, distribution, parallélisation.

KEYWORDS : compilation, distributed memory machines, distribution, parallelization.

1. Introduction

1.1. *Les machines parallèles à mémoire distribuée*

Dans la gamme des ordinateurs offrant une forte puissance de calcul, les machines parallèles à mémoire distribuée figurent en bonne place. Une *machine à mémoire distribuée* se compose d'un ensemble de couples unité de calcul/mémoire locale reliés les uns aux autres par l'intermédiaire d'un réseau d'interconnexion. De tels couples sont généralement désignés sous le nom de *nœuds*. L'unité de calcul de chaque nœud accède directement à sa mémoire locale ; l'accès aux données détenues par les autres nœuds de la machine s'effectue par l'échange de messages. Ces messages circulent sur le réseau d'interconnexion qui peut présenter diverses topologies suivant les machines. Les principales topologies utilisées par les constructeurs sont à caractère régulier : grille, tore, hypercube, etc.

Selon leur mode de contrôle, on peut classer les machines parallèles en deux catégories : les machines SIMD *(Single Instruction Multiple Data),* dont les nœuds effectuent simultanément une même instruction sur les données qui leur sont attribuées, et les machines MIMD *(Multiple Instruction Multiple Data),* dont les nœuds possèdent chacun une unité de contrôle qui gère de façon indépendante le flot des instructions du processus qui lui est dédié.

Nous nous intéressons ici à cette deuxième classe de machines que nous appelons APMD (Architecture Parallèle à Mémoire Distribuée).

1.2. *Le domaine d'application des* APMD

Le type d'applications pouvant être programmées sur les APMD n'est a priori pas limité. Cependant, l'emploi de telles machines est d'autant plus justifié que l'application nécessite un calcul rapide opéré sur d'énormes quantités de données. De plus, pour être un bon candidat pour ce type de machine, une application doit comporter potentiellement un certain parallélisme afin de permettre une distribution efficace des calculs sur les processeurs. Le programme distribué mettant en œuvre l'application doit lui-même respecter certains critères afin d'être exécuté de la manière la plus efficace possible.

Le premier critère est la localité spatiale des accès aux données qui facilite la répartition des calculs et des données sur les processeurs de la machine et évite trop d'accès distants. Le deuxième critère important est la granularité du parallélisme. Un grain moyen semble être idéal pour profiter pleinement des possibilités d'une APMD : les calculs sont suffisamment nombreux par rapport au nombre de communications qui doivent être effectuées.

En conclusion, le cadre de prédilection d'utilisation des APMD est celui des applications appartenant à ce qui est communément appelé le *calcul scientifique*. Une grande partie de ces applications nécessitent des calculs matriciels, calculs qui présentent en général une bonne localité des accès ainsi qu'un parallélisme potentiel de granularité moyenne.

1.3. *La programmation des* APMD

1.3.1. *Approche manuelle*

La première technique employée pour programmer les APMD peut être qualifiée *d'approche manuelle*. Elle consiste à écrire directement les codes d'un ensemble de processus séquentiels gérant chacun leur propre espace mémoire et coopérant par échange de messages. Dans cette approche, le qualificatif *manuel* indique que le programmeur doit prendre en compte à la fois les paramètres de la machine cible et ceux de son application. Les langages utilisés dans ce type d'approche sont soit des langages entièrement dédiés à la programmation par échange de message comme par exemple OCCAM, soit des extensions de langages séquentiels classiques comme C ou Fortran. Dans le second cas, c'est le constructeur qui, pour une machine donnée, fournit des extensions au langage séquentiel qui permettent d'exprimer explicitement la gestion de processus ainsi que l'envoi et la réception de messages.

Quand le nombre de processeurs de la machine utilisée est grand, l'écriture d'un code différent pour chaque processus associé à un processeur devient très difficile. L'utilisateur peut alors avoir recours au modèle de programmation SPMD [DR 85]. SPMD est l'acronyme de "Single Program Multiple Data" ; un algorithme distribué est qualifié de SPMD si tous les processus qui le composent possèdent le même code : l'aspect "distribué" du code se manifestant par l'attribution de données différentes à chaque processus.

Le problème lié à *l'approche manuelle* est la difficulté, pour le programmeur, de gérer la complexité de l'application. Même en adoptant le modèle SPMD, le programmeur ne peut que très rarement maîtriser à la fois la correction et l'efficacité de sa mise en œuvre, lorsqu'il traite de grosses applications. De plus, l'apport de la moindre modification à de telles mises en œuvre nécessite une remise en cause et une vérification délicate du bon fonctionnement du nouveau programme distribué obtenu.

1.3.2. *Approches automatiques*

La seconde technique de programmation des APMD est davantage *automatique* car elle permet au programmeur de s'abstraire partiellement des spécificités de la machine parallèle qu'il souhaite utiliser. Deux types d'abstractions sont possibles : on peut masquer au programmeur l'aspect distribué de la mémoire ou masquer l'aspect parallèle de la machine.

. Dans le premier cas le programmeur conserve la notion de processus parallèles mais possède une vision globale de l'espace mémoire de la machine. La base de cette approche est d'offrir une *mémoire virtuelle partagée* directement accessible par l'ensemble des processus de l'application [LI 89, LAH 92]. Avec cette approche, l'utilisateur peut programmer une APMD comme s'il s'agissait d'une machine parallèle à mémoire commune. Restent donc à sa charge la gestion des processus, leur coopération et leur synchronisation.
. Dans la seconde approche, le principe est d'utiliser une programmation purement séquentielle et un espace de nommage global mais de prendre en compte

la distribution de la mémoire. Ainsi, avec cette seconde approche, les applications écrites par le programmeur ne présentent plus directement la notion de processus. En contrepartie, il est demandé au programmeur un plus grand effort du point de vue de la répartition des données dans l'espace mémoire. En général, l'utilisateur doit préciser pour chaque tableau qu'il souhaite distribuer, une décomposition en groupes d'éléments ; il précise ensuite la répartition de ces groupes dans les mémoires locales liées aux processeurs. Des annotations ou des extensions sont incorporées à un langage séquentiel existant : elles permettent à l'utilisateur d'indiquer les décompositions et les répartitions des données manipulées par son application. Le compilateur se charge ensuite de la distribution de l'algorithme en un ensemble de processus et de la génération des communications nécessaires à l'accès des données [CHA 91, HIR 92, AND 92].

Ces deux approches *automatiques* ont en commun de proposer à l'utilisateur un espace de nommage global. Cependant, avec la première approche, le programmeur considère une localisation *uniforme* des données (mémoire partagée) alors qu'avec la seconde, il prend en compte, dans son programme, l'aspect *non-uniforme* de cette localisation (distribution des données). Pour ce qui est du contrôle, les rôles sont inversés : la première approche en donne une vision répartie (programmation explicite des processus) tandis que la seconde donne la vision d'un contrôle unique (programmation purement séquentielle). Bien que ces deux approches semblent très différentes à première vue, les problèmes que doivent résoudre les concepteurs lors de leur mise en œuvre sont similaires. Ainsi dans les deux cas, il faut essayer de limiter les transferts de données, soit en tentant de faire coïncider les pages virtuelles avec la localité des données dans le cas d'une mémoire virtuelle partagée, soit en regroupant et déplaçant les communications dans le cas de la distribution des données.

A l'heure actuelle les systèmes proposant une approche automatique adoptent un compromis *efficacité du code produit/simplicité de programmation*. L'approche idéale de la programmation des APMD serait celle qui, entièrement automatiquement, génèrerait un code parallèle efficace à partir d'un programme purement séquentiel ne contenant aucune suggestion sur la répartition des données. Mais dans l'état de l'art actuel, la découverte du parallélisme intrinsèque d'une application, sans aide extérieure du programmeur n'est pas encore réalisable. Le compromis adopté n'étant pas très satisfaisant, les recherches se poursuivent dans ce domaine [RAM 91, GUP 92, O'B 92].

2. Compilation pour APMD par distribution des données

Le rôle de ce type de compilation est de passer d'un code séquentiel agissant sur un espace d'adressage global, à un code formé d'un ensemble de processus disposant chacun d'un espace mémoire local. Le schéma de base utilisé par ce genre de compilation s'appuie sur le modèle SPMD et sur le principe des *écritures locales* [CAL 88]. Les processus générés par le compilateur présentent un code générique, mais agissent chacun sur des données qui leur sont propres.

Pour comprendre le principe des *écritures locales*, il faut noter que la spécification d'une distribution établit une relation de *possession* entre les processus et les données

Code original	Code de P1	Code de P2	Code de P3
(1) `b := 10`			`b := 10`
(2) `a := d * 2`	`a := d * 2`		
(3) `c := a + b`	envoyer(`a`) à P2	recevoir(`a`) de P1 recevoir(`b`) de P3 `c := a + b`	envoyer(`b`) à P2

Tableau 1. *Distribution du code sur trois processus*

(éléments de tableau) en associant des parties de tableaux à des processus. Nous dirons qu'un processus *possède* une donnée, si la distribution indique que ce processus stocke la donnée dans sa mémoire locale. Alors, la règle des *écritures locales* impose qu'une instruction du programme source qui modifie une donnée (i.e. une affectation) soit exécutée par le processus qui possède cette donnée.

Prenons un exemple dans lequel, pour simplifier, les données distribuées sont des variables scalaires (et non pas des tableaux), et voyons quel serait le rôle des processus générés en appliquant cette méthode de compilation :

```
ent a, b, c ,d

distribuer a et d sur P1
distribuer b      sur P3
distribuer c      sur P2

b := 10       (1)
a := d * 2    (2)
c := a + b    (3)
```

La compilation de ce pseudo-code génère trois processus P1, P2 et P3 dont les actions effectives sont résumées par le tableau 1 : L'instruction (1) du code séquentiel correspond à l'affectation de la constante 10 à la variable distribuée `b`. Cette variable a été attribuée au processus P3 ; la règle des écritures locales indique donc que c'est P3 qui doit réaliser l'affectation : les deux autres processus P1 et P2 ne sont pas impliqués dans l'exécution de cette instruction.

L'instruction (2) ne va pas non plus entraîner de coopération entre les processus. En effet, la variable lue (`d`) et la variable écrite (`a`) ont été attribuées au même processus (P1). Ce processus peut donc procéder directement à l'affectation qui ne met en jeu que des variables qui lui sont locales. La localité des accès réalisés par les deux instructions (1) et (2), rend possible l'exécution en parallèle des deux affectations.

L'instruction (3) est différente : elle présente des références en lecture aux variables distribuées `a` et `b` qui ont été respectivement attribuées aux processus P1 et P3, et une référence en écriture à la variable `c`, possédée par un troisième processus, le processus P2. P2, qui doit réaliser l'affectation, a besoin d'accéder aux valeurs courantes des variables `a` et `b` qui ne sont pas présentes dans son espace mémoire local. Pour ce faire, il se met en attente de ces valeurs qui sont respectivement envoyées par les processus

P1 et P3. La non-localité des accès entraîne ainsi une coopération des trois processus par échanges de messages.

Dans l'exemple ci-dessus, on connaît statiquement, pour chaque référence, la distribution de la donnée accédée. Ce n'est généralement pas le cas pour un accès indexé, comme par exemple `A[i][j]`, où la valeur des indices n'est connue qu'à l'exécution.

L'impossibilité de connaître statiquement la localisation des données accédées oblige le compilateur à générer un code SPMD composé d'une suite d'instructions *gardées*. L'évaluation de ces gardes à l'exécution, permet à chaque processus de calculer sa contribution à la réalisation d'une instruction. Prenons par exemple l'affectation A[i] := B[j]. Le code SPMD gardé, généré par le compilateur, présenterait la forme suivante :

si *je possède* B[j] *et* A[i]
  alors *exécuter l'affectation* A[i] := B[j]
  sinon **si** *je possède* B[j] *et pas* A[i]
    alors *envoyer* B[j] *au possesseur de* A[i]
    sinon **si** *je possède* A[i]
      alors *recevoir* B[j] *dans* tmp
        exécuter l'affectation A[i] := tmp
      fsi
  fsi
fsi

Il est clair que l'évaluation de toutes ces gardes lors de l'exécution, pénalise les performances du programme. Aussi, un compilateur appliquant ce schéma devra, après une analyse de la localité des accès, optimiser le code en plaçant, quand c'est possible, les gardes au niveau des structures de contrôle englobant les affectations. Par exemple, si pour une boucle donnée, le compilateur est capable de générer la garde à l'extérieur de la boucle, le code sera plus performant.

Un autre facteur à prendre en compte est le coût des opérations de communication qui est très supérieur à celui des opérations arithmétiques. Si le compilateur est capable de minimiser le nombre des communications induites par ce schéma, en regroupant par exemple plusieurs messages ayant même destinataire en une seule opération de communication ou en évitant de répéter des accès distants à des variables n'ayant pas été modifiées, le code produit sera plus efficace.

Enfin, la distribution pose le problème du stockage et de l'adressage des données. Pour les données locales, la taille des tableaux initiaux doit être adaptée afin que chaque processus utilise le moins d'espace mémoire possible. Pour les données distantes véhiculées dans les messages, des zones mémoire allouées temporairement par chaque processus, peuvent être utilisées. Dans les deux cas, l'adressage doit être modifié en conséquence : afin de générer un code performant le compilateur essaiera de réduire le surcoût induit par les calculs mis en jeu par cet adressage.

Si les performances du code produit constituent un critère très important pour juger la qualité d'un compilateur, la largeur du spectre des applications qu'il peut traiter ne doit cependant pas être négligée dans ce jugement. En effet, un compilateur qui imposerait

des restrictions trop fortes sur le langage d'entrée ne peut pas être raisonnablement considéré comme un bon outil de programmation.

Plusieurs travaux de recherche sont menés pour concevoir des environnements de programmation selon ce modèle [ROS 90, QUI 90, KOE 91, CHA 91, AND 92, HIR 92]. En ce qui concerne l'aspect langage, il faut noter l'initiative "High Performance Fortran" [FOR 93] qui vise à normaliser l'expression de la distribution dans le cadre de Fortran. Parmi les travaux en cours, nous avons choisi de présenter trois projets qui traitent de la compilation de programmes séquentiels pour machines à mémoire distribuée et qui s'appuient sur la distribution des données et la règle des écritures locales. Ces trois projets représentatifs de l'approche ont donné lieu à des réalisations de prototypes.

3. Vienna Fortran

Vienna Fortran est la prolongation du projet SUPERB démarré en 1988 par Michael Gerndt et Hans Zima [GER 90]. Ce premier projet a donné naissance à un outil permettant de distribuer, de façon semi-automatique, des programmes séquentiels exprimés en Fortran 77 en vue de leur exécution sur la machine à mémoire distribuée SUPRENUM [GIL 88]. Dans cet environnement, l'utilisateur est invité à préciser, pour chaque unité de programme composant son application, une distribution des tableaux utilisés. Le restructureur procède alors à la génération de processus SPMD exprimés dans le dialecte de Fortran développé pour la programmation parallèle explicite de la machine cible. L'extension de ce système a principalement consisté à définir un langage, Vienna Fortran [CHA 91], construit autour de Fortran 77, permettant d'exprimer directement dans le programme source la distribution des données, puis d'adapter l'outil SUPERB à la compilation de ce nouveau langage [BRE 92].

3.1. *Le langage*

Comme son nom l'indique, la base du langage est Fortran. A ce langage ont été ajoutées des *annotations* de distribution des données ainsi qu'une construction spécifique (le `FORALL`) que nous présentons ici.

3.1.1. *La distribution des données*

La spécification de la distribution des données commence par la définition d'un ensemble de *processeurs virtuels* sur lesquels les éléments des tableaux vont être répartis et les variables scalaires être dupliquées. Il est possible d'imposer une structure à cet ensemble de processeurs par l'intermédiaire de la définition d'un ou plusieurs tableaux de `PROCESSORS`. La première structure définie constitue le *tableau primaire*, les autres définitions ne faisant que fournir une vue différente des processeurs de ce *tableau primaire*. Dans l'exemple ci-après, le tableau `P1` qui constitue le *tableau primaire*, forme une grille de `10x10` processeurs, dont `P2` fournit une vue en ligne.

```
PROCESSORS  P1(10,10)  RESHAPE P2(100)
```

La distribution des données peut alors s'exprimer par rapport à une structure de processeurs, qui par défaut est celle définie par le *tableau primaire*. Il existe deux catégories de distribution : les distributions statiques qui restent valides pour toute la durée de vie des tableaux auxquels elles s'appliquent, et les distributions dynamiques. La catégorie de distribution associée à un tableau est définie à la déclaration de celui-ci. La nature des distributions exprimables en Vienna Fortran est très variée : il est possible de préciser à l'aide de fonctions prédéfinies, des distributions par blocs rectangulaires (*pavés*) répartis en groupes contigüs ou un à un de façon cyclique ; on peut spécifier des distributions irrégulières en utilisant des *tableaux de placements* (*mapping array*) dont les valeurs correspondent à des numéros de processeurs virtuels ; on peut dupliquer certaines dimensions ; on peut aligner la distribution d'un tableau sur celle d'un ou plusieurs autres et enfin, il est même possible pour le programmeur de définir ses propres fonctions de distribution et d'alignement. Voici quelques exemples de distribution statiques et la sémantique qui y est associée :

. distribution *directe*

```
REAL A(10000)    DIST(CYCLIC(10))    TO P1
REAL B(10,10,10) DIST(BLOCK,:,BLOCK) TO P1
```

Les éléments du tableau `A`, groupés par 10, sont distribués de façon cyclique aux processeurs de `P1`. Comme le nombre de dimensions de `A` est inférieur à celui de `P1`, les groupes sont dupliqués sur la deuxième dimension du tableau de processeurs : les processeurs `P1[i][..]` possèdent les mêmes éléments de `A`. Pour le tableau `B`, la première et la troisième dimension sont distribuées en blocs alors que la deuxième dimension n'est pas distribuée : le processeur `P1[i][k]` possède les éléments `B[i][..][k]`.

. distribution *par alignement*

```
REAL C(10,10,10) ALIGN C(K1,K2,K3) WITH B(K2,K1,K3)
```

Dans la distribution spécifiée pour le tableau `C`, les dimensions 1, 2 et 3 sont respectivement alignées sur les dimensions 2, 1 et 3 du tableau `B`. Ainsi dans notre exemple, la première dimension de `C` n'est pas distribuée tandis que les deux dernières dimensions sont distribuées par blocs.

On peut également déclarer des tableaux dont la distribution pourra être modifiée dynamiquement et établir un lien permanent entre les distributions de plusieurs tableaux : la redistribution d'un tableau entraîne alors la redistribution automatique des tableaux connectés.

3.1.2. *La construction* `FORALL`

La construction `FORALL` permet d'indiquer au compilateur qu'une boucle est parallèle : les boucles `FORALL` ne doivent pas comporter de dépendance inter-itération. Aussi, pour compléter le pouvoir d'expression offert par cette construction, les concepteurs de Vienna Fortran ont ajouté la possibilité de spécifier des opérations de réduction

à l'aide d'une boucle FORALL, comme par exemple la somme des éléments d'un tableau, ou toute autre opération qui soit à la fois commutative et associative. Il faut noter que de telles opérations ne peuvent être réalisées avec la sémantique initiale du FORALL car elle comporte des dépendances sur la variable d'accumulation.

Il est également possible de forcer le compilateur à abandonner, pour la compilation d'une boucle FORALL, le principe des écritures locales, en spécifiant l'endroit d'exécution de chaque instance du corps de boucle. Cette spécification peut être donnée soit en terme du possesseur d'une variable distribuée, soit en terme d'un numéro de processeur virtuel. Pour cela, la clause ON est utilisée de la façon suivante :

```
FORALL I=1,N ON OWNER(A(IND(I)))
 A(IND(I)) = (A(IND(I))+B(IND(I))+C(IND(I)))*D(IND(I))
END FORALL

FORALL (I=1,100 , J=1,M,2) ON P2(I)
 T(I,J) = ...
END FORALL
```

L'itération I de la première boucle est exécutée sur le possesseur de A(IND(I)), celle de la seconde boucle est prise en charge par le processeur P2(I).

Cette clause autorise le programmeur à redistribuer le contrôle ; son emploi semble donc réservé aux utilisateurs les plus expérimentés.

3.1.3. *Les procédures*

Le langage Vienna Fortran conserve la notion de procédure (SUBROUTINE) présente dans Fortran 77. Aussi, pour les procédures Vienna Fortran, il est possible de préciser la distribution des tableaux correspondant à des paramètres formels. A l'exécution les paramètres effectifs sont éventuellement redistribués afin de respecter la distribution spécifiée. Les paramètres formels donnent une vision locale des tableaux passés en argument, et la distribution de ces tableaux est propre à la procédure.

Pour éviter la redistribution, il est possible, soit de préciser qu'un paramètre formel hérite de la distribution du paramètre effectif (DIST(*)), soit d'exprimer un test sur la distribution d'un paramètre, qui permet, à l'exécution, de sélectionner un algorithme adapté à chaque distribution. Cette approche est intéressante pour constituer des bibliothèques de fonctions dont le cadre d'utilisation n'est pas connu a priori. Cependant, dans le cas où la distribution est héritée, le compilateur ne dispose pas d'informations suffisantes pour pouvoir optimiser le code de la procédure considérée, et, dans le second cas, la prise en compte de toutes les distributions possibles et l'écriture d'algorithmes adaptés à chacune de celles-ci entraîne une explosion de la taille du code.

3.2. *Le compilateur*

Le compilateur VFCS (*Vienna Fortran Compilation System*) est basé sur le restructureur SUPERB ; il produit, à partir d'un programme source Vienna Fortran, un

code SPMD exprimé dans un dialecte de Fortran possédant la notion de processus et d'échange de messages. Les communications sont exprimées à l'aide des primitives de la bibliothèque PARMACS [BOM 90], bibliothèque ayant été portée sur plusieurs APMD. La tâche du compilateur se répartit en quatre étapes :

1) normalisation

Cette étape consiste en plusieurs analyses (syntaxique, sémantique, de flot de données, de flot de contrôle et de dépendance de données) suivie par une phase de normalisation (bornes de boucles et expressions d'indice).

2) séparation hôte/nœud

Les deux programmes résultant de cette séparation conservent le flot de contrôle du programme initial, mais leurs tâches se répartissent ainsi :

. l'*hôte* est responsable de la gestion des processus *nœuds* et du traitement des entrées/sorties.

. le programme *nœud*, dont le code est de type SPMD, se charge des calculs : dans ce code, les opérations d'entrées/sorties sont remplacées par des échanges de messages avec l'*hôte* (réceptions pour les lectures, et envois pour les écritures).

3) analyse de distribution

Afin de répartir les calculs, la distribution des tableaux doit être connue pour chaque référence présente dans le programme : une analyse inter-procédurale de la distribution est donc nécessaire. Le compilateur VFCS distingue, grâce au graphe d'appels calculé à l'étape de normalisation, les différentes instances des procédures en fonction de la distribution de leur paramètres. Une instance de la procédure est créée (*cloning*) pour chaque combinaison de distribution, et l'appel présent dans le code original est remplacé par un appel à l'instance correspondant à la distribution des paramètres effectifs. Cette distinction permet d'instancier les distributions des paramètres ce qui rend possible l'analyse statique des procédures. Le problème de cette technique est la croissance exponentielle de la taille du code généré qu'elle implique.

4) distribution du code

Le compilateur VFCS traite différemment les boucles `FORALL` des autres instructions. Pour les instructions autres que le `FORALL` la compilation s'effectue en deux étapes : l'application de la règle des écritures locales suivie d'une phase d'optimisation. Les boucles `FORALL` sont traduites en une seule étape suivant la technique de l'*inspector/executor*. Cette technique est décrite dans la section 3.2.2..

3.2.1. *Compilation*

La compilation des instructions situées à l'extérieur des boucles `FORALL` consiste à masquer ces instructions selon le modèle des écritures locales et à insérer les communications nécessaires à la réalisation des accès distants. L'efficacité du code ainsi produit est généralement insuffisante car la communication et le masquage sont traités au niveau de chaque affectation. Une étape d'optimisation du programme nœud est

alors entreprise de façon à réduire le nombre des communications et des masques d'instructions.

L'optimisation du programme nœud se base entièrement sur une *analyse de recouvrement* (overlap analysis). Le but de cette analyse est d'approximer, pour chaque processus nœud et chaque tableau distribué, la partie du tableau à laquelle le processus accède en lecture et qui n'est pas détenue par ce processus. L'analyse de recouvrement, basée sur une technique purement algorithmique, est décrite dans la thèse de Michael Gerndt [GER 90]. L'algorithme utilisé n'est adapté qu'aux distributions dites *standards*, i.e. des distributions pour lesquelles chaque processus possède un et un seul bloc d'un même tableau : il ne peut donc pas s'appliquer aux distributions cycliques.

Pour décrire l'analyse de recouvrement notons A et B deux tableaux distribués de façon standard, p un processeur virtuel, $\vec{i}$ un vecteur d'itération d'un nid de boucles, f et g deux fonctions linéaires de $\vec{i}$, et Loc_p^A le bloc local de A associé à p par la distribution. Pour chaque référence $B[g(\vec{i})]$ située en partie droite d'une affectation de membre gauche $A[f(\vec{i})]$, et pour chaque processeur p, on commence par calculer un sur-ensemble (pavé englobant) des vecteurs d'indices susceptibles[1] de générer une écriture sur Loc_p^A. L'image de ce pavé par la fonction g, Lu_p, caractérise un sur-ensemble des données susceptibles d'être lues par le processeur p. Le pavé $Englob_p$ qui englobe Loc_p^B et Lu_p est calculé. On en déduit le descripteur spécifique à p, $Drec_p^{B[g(\vec{i})]}$, qui caractérise pour chaque dimension du bloc, les deux extensions (aux extrémités) nécessaires pour passer de Loc_p^B à $Englob_p$. Le *descripteur de recouvrement* associé à la référence $B[g(\vec{i})]$, $Drec^{B[g(\vec{i})]}$, est obtenu par fusion des $Drec_p^{B[g(\vec{i})]}$.

Le compilateur VFCS utilise les *descripteurs de recouvrement* à deux fins : la réservation mémoire associée aux tableaux distribués et l'optimisation des communications.

Afin d'obtenir un code SPMD, le compilateur réserve pour chaque tableau distribué B, la partition locale étendue grâce à l'union des descripteurs de recouvrement de chaque référence à B : cette extension définit la *zone de recouvrement* associée au tableau B. Ceci simplifie l'allocation mémoire et l'adressage en regroupant dans une même zone les données locales et distantes. Cependant, la suite d'approximations réalisées peut conduire à déclarer sur chaque processus, une zone de taille très supérieure à celle du domaine de données auquel le processus accède réellement. Dans le pire des cas, l'intégralité du tableau initial est réservée pour chaque processus. La figure 1 illustre un cas favorable pour lequel le nombre d'éléments réservés mais inutilisés est faible : seule la réservation des quatre éléments de coin est superflue.

L'instruction EXSR($B[g(\vec{i})], Drec^{B[g(\vec{i})]}$) est utilisée pour traduire les références en lecture aux tableaux distribués. L'analyse de dépendances réalisée à l'étape de normalisation, sert au compilateur à vérifier s'il est possible de déplacer certaines instructions EXSR à l'extérieur des boucles.

Quand l'extraction est possible, l'exécution sur chaque processeur de l'instruction EXSR, consiste à déterminer les sections de tableaux à échanger avec les autres processeurs d'après les descripteurs de recouvrement et le volume décrit par la référence.

[1] *susceptibles* car les bornes de boucles ne sont pas toujours prises en compte dans le calcul

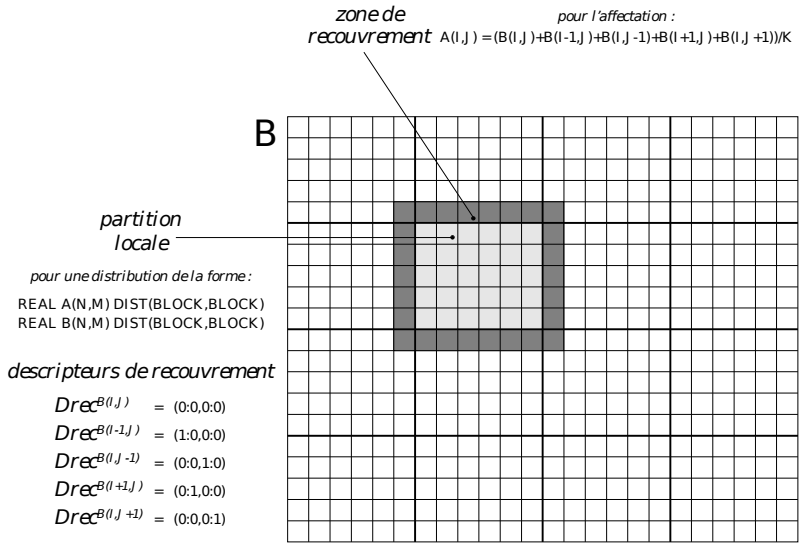

Figure 1. *Analyse de recouvrement*

Les communications sont alors vectorisées de façon à transporter dans un message non plus un seul élément de tableau, mais toute une section.

La seconde partie des optimisations réalisées par le compilateur concerne le masquage. Elle consiste à décomposer l'espace d'itération d'une boucle par une méthode de *strip-mining* [WOL 89] de façon à réduire, quand c'est possible, l'espace parcouru par chaque processeur à la zone dont il a effectivement la charge. Au niveau du code du programme nœud, cette décomposition se traduit en général par une modification de la valeur des bornes de la boucle initiale : les nouvelles valeurs sont fonction de l'identité du processeur qui exécute le code.

3.2.2. *Compilation des boucles* FORALL

Pour le traitement des boucles FORALL, le système Vienna Fortran adopte une technique d'analyse réalisée à l'exécution, connue sous le nom d'*inspecteur/exécuteur* : cette technique a été développée dans le cadre du projet Kali [KOE 91]. Elle est mise en œuvre dans le compilateur VFCS à l'aide de la bibliothèque PARTI [DAS 92] dédiée à la gestion de données distribuées. Cette bibliothèque a été développée par Joel Saltz au cours de ses travaux visant l'exécution, sur machines à mémoire distribuée, de programmes agissant sur des structures de données irrégulières [BER 90].

Le compilateur génère les codes des phases d'inspection et d'exécution. Durant la phase d'inspection, chaque processeur exécute une version simplifiée de la boucle puisqu'il n'examine que les références aux variables distribuées pour constituer :

. la liste des itérations *purement* locales (écriture et lecture de données locales)

. la liste des autres itérations locales (lecture de données distantes)
. la liste des données à émettre
. la liste des données à recevoir

La phase d'exécution parcourt ces listes pour réaliser dans l'ordre : les émissions, les calculs *purement* locaux, les réceptions et les calculs utilisant les données reçues.

La technique de l'inspecteur/exécuteur engendre un surcoût très important à la fois en mémoire (stockage des listes) et en temps d'exécution (la complexité de la phase d'inspection est la même que celle de la boucle séquentielle).

4. Fortran D

Le projet Fortran D [HIR 91][2] est dirigé par Ken Kennedy, qui est le premier a avoir défini la transformation de programmes séquentiels par l'application de la règle des écritures locales et le respect du modèle SPMD [CAL 88]. Le compilateur actuellement mis en œuvre par ce projet [HIR 92, TSE 93], réalise la transformation de programmes séquentiels (écrits dans le langage Fortran D) comportant des spécifications de décomposition et de distribution de tableaux, en programmes explicitement parallèles exprimés dans le dialecte de Fortran développé par Intel pour la programmation de l'iPSC/860.

4.1. *Le langage*

L'approche adoptée par Fortran D pour la distribution des tableaux diffère de celle choisie par Vienna Fortran : elle comprend en effet, un niveau d'abstraction supplémentaire. Avant d'être distribués, les tableaux de données sont alignés sur des tableaux virtuels appelés *décompositions*, représentant un espace d'indices[3]. Le choix de ces alignements ne dépend que de la structure de l'algorithme et est ainsi indépendant de l'organisation et du nombre des processeurs qui exécuteront les calculs décrits par l'algorithme. Une fois l'alignement précisé, la distribution des *décompositions* est spécifiée. C'est ici que les caractéristiques de la machine peuvent être prises en compte. La localisation des données est donc fonction à la fois de la distribution des *décompositions* et de l'alignement des tableaux sur ces *décompositions.*

4.1.1. *L'alignement des tableaux*

Outre l'alignement *exact*, le langage Fortran D propose notamment les alignements *décalé*, *effondré* et *dupliqué*, illustrés par la figure 2. La liste exhaustive des alignements exprimables peut être trouvée dans la thèse de Chau-Wen Tseng [TSE 93].

---

[2] Le "D" de Fortran D fait référence aux mots *Données, Décomposition* et *Distribution*

[3] Ce concept est voisin de celui de VIEW, offert par le langage BOOSTER [PAA 90]

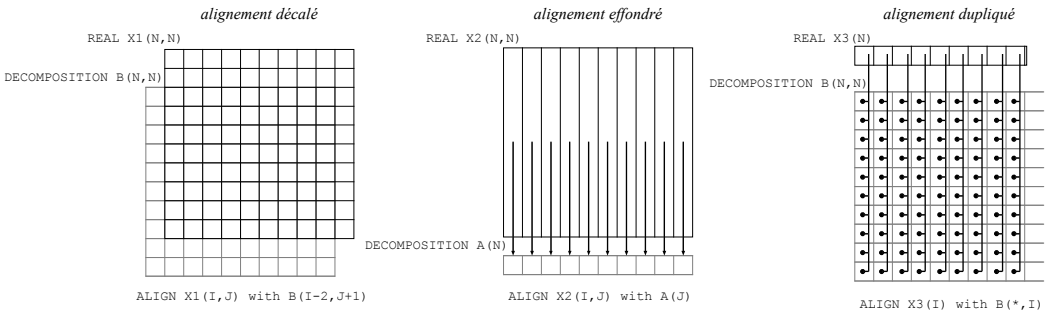

Figure 2. *Exemples d'alignements exprimables en Fortran D*

4.1.2. *Distribution*

Après avoir spécifié les alignements des tableaux sur des *décompositions*, le programmeur précise une distribution pour chaque *décomposition*, comme par exemple :

```
DISTRIBUTE A(BLOCK(8)), B(BLOCK(4),CYCLIC(2))
DISTRIBUTE B(BLOCK_CYCLIC(2,8),*)
```

La distribution `BLOCK(P)` répartit la dimension (de taille `N`) sur `P` processeurs en tranches de taille `N/P`. La distribution `CYCLIC(P)` répartit la dimension en tranches de taille `1` en procédant de manière cyclique : le processeur `i` se voit attribuer les tranches `i`, `i+P`, `i+2P`, etc. La distribution `BLOCK_CYCLIC(T,P)` est identique à la distribution `CYCLIC(P)` mise à part la taille des tranches qui n'est plus `1` mais `T`. Enfin, pour préciser qu'une dimension n'est pas distribuée, le symbole `*` est utilisé. Ainsi la distribution `DISTRIBUTE B(BLOCK_CYCLIC(2,8),*)` répartit de façon cyclique parmi `8` processeurs, des groupes de `2` lignes.

Comme en Vienna Fortran, il est possible de spécifier des distributions irrégulières en passant par l'intermédiaire d'un *tableau de placement*.

Il est important de noter que les spécifications d'alignement et de distribution se font par l'intermédiaire d'instructions et non pas de déclarations, ce qui rend possibles le réalignement et la redistribution des tableaux au cours d'un programme.

4.1.3. *La construction* `FORALL`

La construction `FORALL` offerte par le langage Fortran D permet d'exprimer le parallélisme d'un groupe d'affectations. Elle diffère du `FORALL` de Vienna Fortran de part sa sémantique dite COPY-IN, COPY-OUT qui se définit ainsi : chaque instance du corps de boucle travaille sur une copie de l'espace des données et ne peut donc utiliser que les valeurs définies par cette instance ou celles définies avant la boucle. Comparons les deux boucles suivantes :

```
DO I = 2,N                    FORALL I = 2,N
   X(I) = X(I-1)                 X(I) = X(I-1)
END DO                        END FORALL
```

Après l'exécution de la version *séquentielle* DO, tous les éléments du tableau X valent X[1], alors que la version *simultanée* FORALL, opère en parallèle un décalage de X. Cette construction permet d'exprimer des opérations globales sur les tableaux similaires à celles proposées par Fortran 90. Les itérations qu'elle décrit peuvent être compilées efficacement sans analyse de dépendance de données sophistiquée.

Comme en Vienna Fortran, la clause on peut être utilisée pour préciser l'attribution des itérations aux processeurs; la spécification d'opérations de réduction est également possible.

4.1.4. *Les procédures*

Dans le cadre de l'utilisation de procédures, le langage Fortran D adopte les conventions suivantes :

1) Les tableaux constituant les paramètres formels d'une procédure héritent de la distribution des paramètres effectifs. Cette distribution par défaut est valide jusqu'à la rencontre d'une instruction de distribution.
2) Un tableau distribué, passé en argument d'une procédure, retrouve, au retour de la procédure sa distribution d'avant l'appel, même si la procédure modifie la distribution de ses paramètres.

De plus, le compilateur Fortran D interdit tout appel de procédure à l'intérieur d'une boucle FORALL. En effet, chaque instance d'une itération d'un FORALL est destinée à être exécutée sur un processeur unique et ce, de façon indépendante. Dans ce cadre d'exécution indépendante, la coopération entre les processeurs nécessaire à l'exécution de toute procédure, est irréalisable.

4.2. *Le compilateur*

Le compilateur Fortran D est développé dans le cadre de ParaScope [BAL 89], un environnement de programmation parallèle permettant l'analyse et la transformation de programmes. Le compilateur procède en trois phases que nous allons décrire une à une.

4.2.1. *Phase d'analyse*

Cette phase commence par une série d'analyses classiques (analyse de dépendances de données, propagation de constantes, reconnaissance des invariants de boucle, etc.) permettant à la fois la simplification du programme analysé et la constitution d'informations utilisées par les phases suivantes du processus de compilation.

Pour chaque distribution d'un tableau A, le compilateur calcule la *fonction de distribution*, $\delta_A(\vec{i})$, qui à partir du vecteur d'indices d'un élément de A donne le numéro du possesseur de l'élément. Dans la version actuelle du compilateur Fortran D, l'analyse qui consiste à déterminer pour chaque accès la distribution associée au tableau référencé se trouve simplifiée par les restrictions imposées à l'écriture des procédures. En effet, toute procédure doit impérativement commencer par définir une distribution

locale de tous ses paramètres : la tâche d'analyse de distribution est donc réduite à une analyse locale à chaque procédure car l'héritage de distribution n'est pas pris en compte par le compilateur.

À l'aide de la fonction δ_A, le compilateur détermine l'ensemble des éléments de A associés au processeur p, appelé *ensemble des indices locaux*[4] : $image_A(p) = \{\vec{i} \mid \delta_A(\vec{i}) = p\}$. Ensuite, pour chaque référence $A(f(\vec{k}))$ apparaissant en partie gauche d'une affectation située dans un nid de boucles de vecteur d'itération $\vec{k}$ et de domaine $\mathcal{D}$, on calcule l'*ensemble des itérations locales* au processeur p : $IL_p^{A(f(\vec{k}))} = f^{-1}(image_A(p)) \cap \mathcal{D}$. Il caractérise l'ensemble des itérations pour lesquelles la référence $A(f(\vec{k}))$ est locale au processeur p. Pour une affectation, l'ensemble des itérations locales est défini comme étant l'ensemble des itérations locales associé à la référence située en partie gauche.

Afin de préparer la répartition des calculs entre les processeurs, le compilateur, pour chaque nid de boucles présent dans le programme, classe les affectations en *groupes* : les affectations qui possèdent le même ensemble d'itérations locales sont classées dans le même *groupe*. Si à la fin de ce classement, un seul *groupe* est obtenu, le nid correspondant est dit *uniforme*. La distribution des calculs décrits par un tel nid peut être entièrement réalisée à la compilation en réduisant les bornes des boucles afin que celles-ci ne décrivent que les itérations locales à chaque processeur. Le traitement des nids *non-uniformes* nécessite l'insertion de gardes à l'intérieur du nid de boucles : à l'exécution chaque processeur évalue ces gardes et détermine ainsi s'il est responsable de l'exécution des affectations correspondantes.

Après avoir effectué le classement des affectations d'un nid de boucles de vecteur d'itération $\vec{k}$ et de domaine $\mathcal{D}$, le compilateur procède à l'évaluation des accès distants mis en jeu. Pour chaque processeur p et pour chaque référence $B(g(\vec{k}))$ en partie droite d'une affectation de membre gauche $A(f(\vec{k}))$, le compilateur calcule $accès_p^{B(g(\vec{k}))}$ qui définit l'ensemble des itérations pour lesquelles p accède un élément de B. Pour cela il applique la fonction d'accès g à l'ensemble des itérations locales associé à l'affectation et intersecte le résultat avec le domaine d'itération : $accès_p^{B(g(\vec{k}))} = g(IL_p^{A(f(\vec{k}))}) \cap \mathcal{D}$. Enfin, pour obtenir l'ensemble des itérations pour lesquelles la référence $B(g(\vec{k}))$ représente un accès distant pour p, le compilateur soustrait $IL_p^{A(f(\vec{k}))}$ à $accès_p^{B(g(\vec{k}))}$. Le calcul de ces ensembles permet d'identifier les communications nécessaires à l'évaluation de chaque affectation.

Le compilateur utilise une structure de données appelée *Descripteur de Section Régulière* (DSR) pour approximer les ensembles d'itérations calculés. Les DSR offrent une représentation compacte de domaines triangulaire-rectangles [BAL 90]. Dans la mise en œuvre actuelle du compilateur, seuls les domaines rectangulaires sont pris en compte ; une représentation sous forme de vecteurs de doublets (borne inférieure, borne supérieure) est utilisée. Cette restriction simplifie beaucoup les calculs d'union, d'intersection et de différence de DSR qui sont très souvent effectués par le compilateur dans le cadre de l'optimisation du code.

[4] Ceci définit en fait la *partition locale* à p du tableau A.

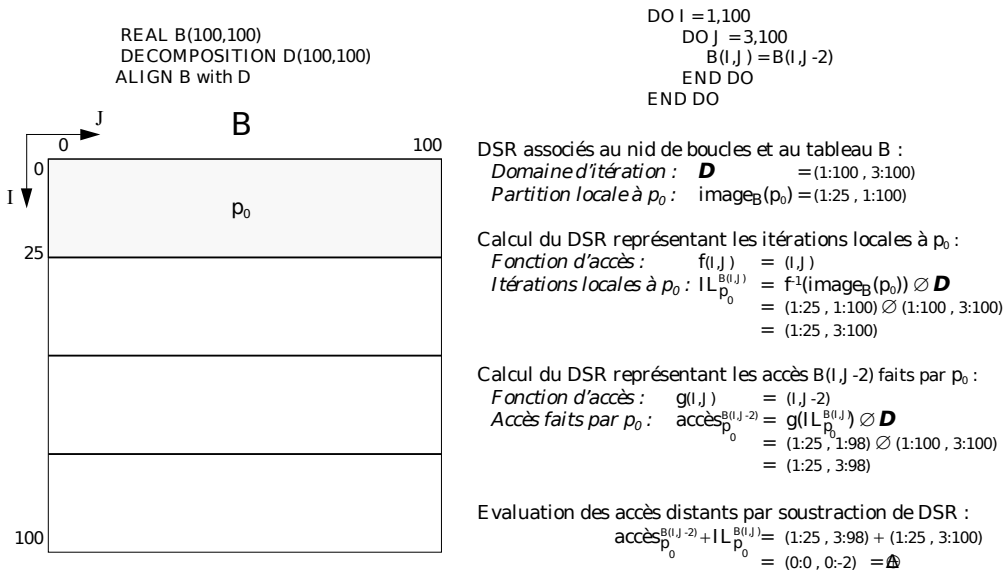

Figure 3. *Exemple de calculs sur les DSR*

En général, la réalisation de ces opérations se résume à calculer des maxima et des minima entre les bornes des domaines décrits. La figure 3 illustre l'emploi des DSR pour l'évaluation des accès distants.

4.2.2. *Phase d'optimisation*

Le compilateur réalise deux sortes d'optimisations : celles qui visent à réduire le surcoût engendré par les communications et celles qui visent à augmenter le parallélisme.

La première gamme d'optimisations comprend la vectorisation des messages (extraction des communications hors des boucles puis réorganisation), l'unification des messages (suppression des communications redondantes), le regroupement des messages (fusion des messages ayant un destinataire commun), la détection des communications globales (utilisation des primitives de diffusion et de regroupement), le déplacement des messages vectorisés (les envois sont faits au plus tôt et les réceptions au plus tard, afin d'effectuer en parallèle les calculs et les communications).

La deuxième gamme d'optimisations est basée sur des techniques de transformation de programmes et notamment des boucles. Les transformations réalisées sont l'échange de boucles (sépare les itérations entièrement locales des autres itérations, ce qui permet de placer ces calculs locaux entre les envois de messages et les réceptions de messages nécessaires à la réalisation des calculs non-locaux), la distribution de boucles (crée une suite de boucles parallèles composées d'une seule instruction donc uniformes et donc compilées sans insertion de gardes), le *strip-mining* (découpe les boucles

en groupes d'itérations, pour lesquels les besoins en mémoire pour le stockage des valeurs distantes accédées est réduit d'autant), ainsi que la fusion de boucles (utilisée principalement pour transformer les nids de boucles non-parfaitement imbriqués de façon à rendre applicables l'échange de boucles et le strip-mining).

4.2.3. *Phase de génération de code*

Cette phase commence par l'insertion de code d'initialisation : le nombre de processeurs `n$proc` et l'identité de chaque processeur `my$proc` sont fixés. Les partitions locales sont ensuite instanciées : les bornes des tableaux sont ajustées au vu de la distribution. Les bornes des boucles des nids *uniformes* sont calculées en fonction de l'identité de chaque processeur. Pour les autres nids de boucles, les bornes sont inchangées, des zones de mémoire temporaires sont allouées et les gardes adéquates sont insérées.

Pour chaque distribution d'un tableau A, le compilateur calcule la fonction $\alpha_A(\vec{i})$ qui transforme un vecteur global d'indices d'un élément de A en un vecteur d'indices d'accès à la partition locale. Les indices des références tableau sont transformés grâce à ces fonctions. Les fonctions inverses sont également calculées afin de pouvoir traiter les références aux variables d'itération qui ne font pas partie d'une expression d'indice et qui par conséquent doivent conserver leur valeur globale.

Afin de satisfaire les accès distants, des appels aux primitives de communication du système NX/2 de l'iPSC/860 sont insérés dans le programme. Les envois de messages sont non-bloquants et les réceptions sont bloquantes.

Enfin, le type de stockage des données distantes est sélectionné en fonction des calculs traités : une zone de recouvrement est déclarée si les calculs sont réguliers et très locaux, sinon des tampons mémoires sont utilisés. Les références dans les expressions sont changées en conséquence : dans le cas de l'utilisation d'une zone de recouvrement seule l'adaptation des indices des tableaux est nécessaire, dans l'autre cas les accès aux tableaux sont transformés en accès aux tampons mémoires.

5. Pandore II

L'environnement de programmation Pandore II réalisé à l'IRISA [AND 90, AND 92, AND 93] comprend un compilateur, des supports d'exécution pour différentes APMD et des outils d'analyse de performances. Le langage source est syntaxiquement inspiré du langage C, mais le compilateur pourrait être facilement adapté à d'autres langages impératifs comme par exemple Fortran.

5.1. *Le langage C-Pandore II*

5.1.1. *Présentation générale du langage*

Le langage C-Pandore II est basé sur le langage C. Il reprend la même syntaxe et conserve les constructions que C et d'autres langages impératifs séquentiels ont en

```
#define N    1024
#define P       4     /* nombre de processeurs */
#define W       1.5

float M[N][N];

float norm(float Enorth,float Esouth,float Ewest,float Eeast,float E)
{
 return ((W/4) * (Enorth + Esouth + Ewest + Eeast) + E  * (1-W));
}

dist relax(float A[N][N] by block(N/P,N) map wrapped(0,1) mode INOUT)
{int i,j,k;

 for (k=0; k<4; k++)
   for (i=1; i<(N-1)/2; i++)
     for (j=1; j<(N-1)/2; j++)
       A[2*i][2*j] = norm(A[2*i-1][2*j], A[2*i+1][2*j],
                          A[2*i][2*j-1], A[2*i][2*j+1], A[2*i][2*j]);
}

main()
{/* initialisation de M */
 relax(M);
}
```

Figure 4. *Exemple de programme C-Pandore II*

commun, à savoir l'affectation, la conditionnelle, l'itération, la structure de blocs, la déclaration de fonction. Les constructeurs de type pointeur, structure et union ne sont pas disponibles. En effet, le traitement des pointeurs dans un espace mémoire distribué pose le problème de l'identification du possesseur de la zone mémoire pointée. Quant à l'ajout des constructeurs structure et union, il nécessiterait la définition d'une nouvelle expression de distribution pour les objets composites ainsi construits.

Le langage C-Pandore II contient la notion de fonction, mais se limite au sous-ensemble formé des fonctions qualifiées de *closes* (sans effet de bord) que le processus appelant peut exécuter de manière autonome.

Le langage C-Pandore II regroupe toutes les informations relatives à la répartition des données dans une seule et même construction : la *phase distribuée* qui précise comment sont réparties et utilisées les données qui lui sont passées en paramètre. Elle indique également la distribution des tableaux qui lui sont locaux. Le style de programmation lié à l'utilisation du concept de phase distribuée est à rapprocher de la notion de procédurage. Un programme C-Pandore II se compose d'un ensemble de déclarations de phases distribuées suivi de la déclaration d'un programme principal. Ce dernier contient une suite d'instructions séquentielles parmi lesquelles apparaissent des appels aux différentes phases distribuées précédemment déclarées.

La *phase distribuée* se présente comme une procédure pour laquelle, à chaque paramètre, est associée une spécification de répartition. Cette construction permet d'identifier les phases de calcul devant être distribuées par le compilateur. Le mode de passage de paramètres est le *passage par adresse.* Une phase distribuée ne peut être appelée que depuis le programme principal. En particulier, on ne peut pas actuellement imbriquer les appels.

5.1.2. *Les phases distribuées*

Le cœur du langage C-Pandore II est la construction `dist` qui est utilisée pour déclarer une phase distribuée. Il existe trois classes de paramètres pour une phase distribuée : les tableaux distribués, les tableaux dupliqués et les scalaires qui sont systématiquement dupliqués sur chaque processus.

Tout paramètre d'une phase distribuée se voit associer un mode qui précise si la valeur du paramètre effectif est significative à l'entrée de la phase (IN), à la sortie (OUT) ou à l'entrée et à la sortie (INOUT). L'indication de ce mode guide le compilateur dans la génération des communications entre le processus maître et les processus esclaves.

Il est possible de déclarer des variables distribuées dans l'en-tête de phase dont la portée est limitée au corps de la phase.

Un tableau distribué est décomposé en *blocs* de tailles égales qui sont associés aux processus. Pour un tableau distribué donné, l'ensemble des *blocs* associés à un processus constitue la *partition locale* à ce processus. Une spécification de distribution d'un tableau d'entiers V à la forme suivante :

$$int\ V[h_0]...[h_{n-1}]\ by\ block\ (s_0, ..., s_{n-1})\ map \left| \begin{array}{l} regular \\ wrapped \end{array} \right| (d_0, ..., d_{n-1})$$

Les deux étapes de la distribution sont respectivement spécifiées par la fonction de *décomposition* des dimensions du tableau ($block$) et la fonction de *placement* qui associe les blocs aux processus ($regular$ ou $wrapped$).

La fonction $block$ indique la taille de chacune des dimensions d'un bloc de données. Elle ne permet donc d'exprimer que des décompositions par blocs rectangulaires : d'autres types de décompositions, comme par exemple la constitution de blocs suivant une diagonale, ne sont pas exprimables en C-Pandore II.

Les fonctions $regular$ et $wrapped$ précisent sur quels processus sont placés les blocs de données. La fonction $regular$ constitue des groupes de blocs contigüs en fonction du nombre de processus indiqué par le programmeur et du nombre de blocs générés par la fonction de *décomposition* ; elle place ensuite chaque groupe sur un processus différent. La fonction $wrapped$ associe les blocs aux processus un par un, en procédant de manière cyclique ; elle permet ainsi de placer sur un même processus des blocs de données qui ne sont pas contigüs dans le tableau original. La figure 5 illustre la *décomposition* et le *placement* d'un tableau bidimensionnel ainsi que la numérotation des blocs de données.

5.2. *Le compilateur*

5.2.1. *Le schéma de compilation général*

Le compilateur Pandore II génère, à partir d'un programme exprimé dans un langage séquentiel agissant dans un espace de nommage global, un ensemble de processus parallèles destinés à s'exécuter sur une APMD. Le *modèle de programme* correspondant au code généré par le compilateur est le modèle *Maître/Esclave*. Le processus *Maître* contrôle l'enchaînement des phases distribuées ainsi que les entrées/sorties.

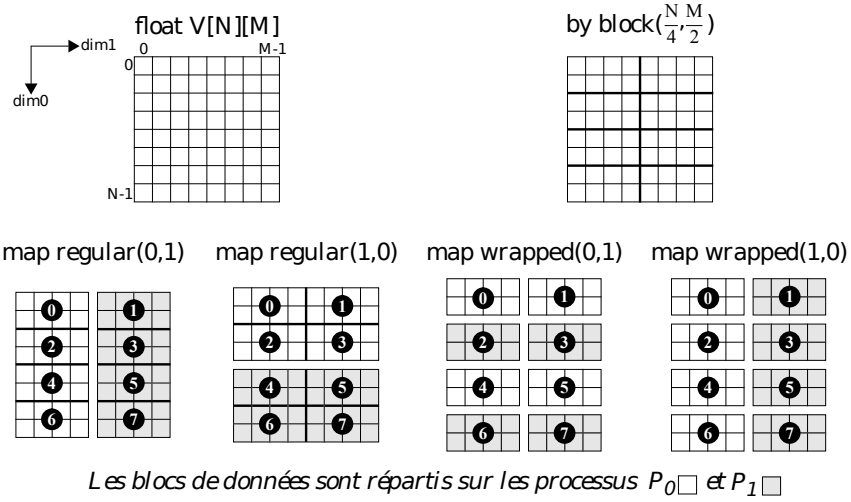

Figure 5. *Décomposition et placement des blocs de données*

Les processus *Esclaves* se chargent d'effectuer le calcul décrit par le corps des phases distribuées.

La distribution du code séquentiel initial est guidée par la répartition des données : le respect du principe des *écritures locales* dicte l'assignation des instructions aux processus *Esclaves.* Ces processus sont de type SPMD, c'est à dire qu'ils sont munis d'un même code, mais qu'ils agissent sur des données différentes. Afin de générer un code unique pour ces processus, le compilateur fait dépendre chaque opération de communication et d'affectation de la possession des variables concernées.

5.2.2. *Le schéma de base : Refresh/Exec/Free*

Nous décrivons ici le schéma de traduction de l'instruction de base qu'est l'affectation, introduit dans [AND 93]. Dans cette description, tous les ensembles considérés sont des ensembles ordonnés ; ils peuvent être parcourus dans l'ordre grâce à la structure de contrôle **pourtout**. Pour une instruction $\mathcal{I}$ et un ensemble $\mathcal{R}$ de références à des variables distribuées, nous définissons les trois ensembles suivants :

. USE($\mathcal{I}$) désigne l'ensemble des références en *lecture* à des variables *distribuées* présentes dans l'instruction $\mathcal{I}$.
. DEF($\mathcal{I}$) désigne l'ensemble des références en *écriture* à des variables *distribuées* présentes dans l'instruction $\mathcal{I}$.
. OWN($\mathcal{R}$) désigne l'ensemble des processus qui *possèdent* les variables atteintes par les références présentes dans $\mathcal{R}$.

Prenons par exemple l'instruction $\mathcal{I}$ suivante :

$$\mathcal{I} \equiv A[j] = B[j+1] * C[j-1]$$

où les tableaux A, B et C sont distribués et la variable entière j est, de par sa nature scalaire, dupliquée. Les références à des variables distribuées sont donc $A[j]$, $B[j+1]$ et $C[j-1]$. Les ensembles $USE(\mathcal{I})$ et $DEF(\mathcal{I})$ prennent pour valeurs :

$$\begin{aligned} USE(\mathcal{I}) &= \{B[j+1], C[j-1]\} \\ DEF(\mathcal{I}) &= \{A[j]\} \end{aligned}$$

Pour la déclaration "`int A[100] by block(10) map regular(0)`", l'élément $A[9]$ réside sur le processus 0. Ainsi pour $j = 9$:

$$OWN(\{A[j]\}) = \{0\}$$

Le schéma de base consiste à traduire l'instruction $\mathcal{I}$ par une **séquence** de trois opérations qui sont :

. Refresh($\mathcal{M}$,USE($\mathcal{I}$),OWN(DEF($\mathcal{I}$)))
. Exec(OWN(DEF($\mathcal{I}$)),$\mathcal{I}/_{\ell \in \text{USE}(\mathcal{I}) \rightarrow m_\ell}$)
. Free($\mathcal{M}$)

Dans ces trois opérations $\mathcal{M}$ désigne un ensemble de variables temporaires allouées pour la réception de valeurs distantes. Pour une référence en lecture, ℓ, à une variable distribuée, la variable, dans laquelle la valeur référencée est stockée, est notée m_ℓ.

L'opération Refresh($\mathcal{M}, \mathcal{R}, \mathcal{P}$) a pour effet de rendre accessibles, localement aux processus de $\mathcal{P}$, les valeurs des variables distribuées référencées dans $\mathcal{R}$. Ces valeurs sont stockées sur les processus de $\mathcal{P}$ dans l'espace mémoire temporaire désigné par $\mathcal{M}$.

L'opération Exec($\mathcal{P}, \mathcal{I}/_{\ell \in \text{USE}(\mathcal{I}) \rightarrow m_\ell}$) fait en sorte que seuls les processus appartenant à $\mathcal{P}$ exécutent l'instruction $\mathcal{I}/_{\ell \in \text{USE}(\mathcal{I}) \rightarrow m_\ell}$. La notation $\mathcal{I}/_{\ell \in \text{USE}(\mathcal{I}) \rightarrow m_\ell}$ désigne l'instruction $\mathcal{I}$ dans laquelle les références en lecture, ℓ, à des variables distribuées distantes ont été substituées par les références aux variables temporaires, m_ℓ, associées. L'espace mémoire $\mathcal{M}$ occupé par ces variables temporaires est réservé par l'opération Refresh($\mathcal{M}, \mathcal{R}, \mathcal{P}$).

Enfin, l'exécution de l'opération Free($\mathcal{M}$) conduit à la libération de la zone mémoire allouée pour $\mathcal{M}$.

Il est à noter qu'une version étendue de ce formalisme a été développée [THO 92]. Elle offre une façon plus générale de décrire les accès aux données et les calculs dans les programmes SPMD, permettant ainsi de s'abstraire de la règle des écritures locales. Cependant, la puissance d'expression de ce formalisme étendu n'est pas nécessaire pour la description du schéma adopté, et nous nous tiendrons donc à l'utilisation de la version simple.

Voyons maintenant comment s'exprime, pour chaque processus SPMD, l'opération Refresh en termes d'échange de messages. Indiquons tout d'abord le cadre dans lequel un processus va s'exécuter. Dans le *modèle de machine d'exécution* retenu, chaque processus peut communiquer avec n'importe quel autre processus par l'intermédiaire d'un canal de communication bi-points. Les communications sur ces canaux sont supposées fiables et FIFO, c'est à dire sans perte, ni déséquencement, ni duplication de messages. Les primitives de communication offertes par le modèle sont les suivantes :

. SEND($\mathcal{D}, v$) exprime l'envoi de la valeur v à tous les processus appartenant à l'ensemble $\mathcal{D}$ des *destinataires*. Un processus qui active une primitive SEND n'est pas bloqué.
. RECV(E, t) : exprime la réception dans une variable temporaire t, d'une valeur en provenance d'un processus *émetteur* E. Un processus qui exécute la primitive RECV est bloqué jusqu'à la réception effective de la valeur.

En plus de ces primitives, le modèle définit une constante, `myself`, qui est propre à chaque processus et dont la valeur correspond à l'identité unique associée au processus. Les éléments de l'ensemble $\mathcal{D}$ ainsi que la valeur E mentionnés ci-avant sont des identités de processus de même nature que la constante `myself`.

À l'aide de cette constante et de ces primitives de communication, l'opération Refresh peut s'exprimer ainsi :

Refresh($\mathcal{M}, \mathcal{U}, \mathcal{P}$) $\equiv$ $\mathcal{M} := \emptyset$
pourtout $v \in \mathcal{U}$ **faire**
$\mathcal{M} := \mathcal{M} \bigcup$ *allouer(t)*;
`refreshvar`$(t, v, \mathcal{P})$
fin-pourtout

Après avoir alloué l'espace mémoire nécessaire à la réception de messages, toutes les variables distribuées distantes sont *rafraîchies*. Un *rafraîchissement* élémentaire consiste en l'exécution du code suivant :

`refreshvar`$(t, v, \mathcal{P})$ $\equiv$ **si** `myself` $\in$ OWN($\{v\}$)
alors SEND($\mathcal{P} \backslash$OWN($\{v\}$), v)
si `myself` $\in \mathcal{P} \backslash$OWN($\{v\}$)
alors RECV(OWN($\{v\}$),t)
si `myself` $\in \mathcal{P} \bigcap$OWN($\{v\}$)
alors $t := v$

À l'aide de la première conditionnelle de ce rafraîchissement élémentaire, chaque processus teste s'il possède la variable à rafraîchir, auquel cas, il envoie la valeur de cette variable à tous les processus de $\mathcal{P}$ qui ne la possèdent pas. En exécutant la deuxième conditionnelle, chaque processus teste son appartenance à l'ensemble des processus de $\mathcal{P}$ qui ne possèdent pas la variable v. Si le processus appartient à cet ensemble, il attend de recevoir la valeur de v, en provenance du processus qui possède cette variable. À la réception, il stocke la valeur de v dans le temporaire t. La troisième conditionnelle conduit les processus de $\mathcal{P}$ qui possèdent la variable à rafraîchir à recopier la valeur de cette variable dans le temporaire t.

La constante `myself` permettant de différencier les processus, la traduction de l'opération Exec s'exprime aisément à l'aide d'un test sur la valeur de cette constante :

Exec($\mathcal{P}, \mathcal{I}$) $\equiv$ **si** `myself` $\in \mathcal{P}$
alors $\mathcal{I}$

La traduction de l'opération Free est la suivante :

$$\begin{array}{lll} \text{Free}(\mathcal{M}) & \equiv & \textbf{pourtout } t \in \mathcal{M} \textbf{ faire} \\ & & \quad \textit{libérer(t)} \\ & & \textbf{fin-pourtout} \end{array}$$

Une gestion mémoire plus subtile peut être envisagée, avec laquelle l'allocation des temporaires est prise en charge par l'opération `refreshvar` au niveau de la deuxième conditionnelle : la réservation mémoire n'est alors faite que si cela est nécessaire. Mais dans ce cas, les opérations Refresh, Exec et Free doivent être intégrées dans une seule et même construction afin de différencier, dans la partie droite de l'affectation, les références aux variables suivant que le processus y accède directement ou via un temporaire : ce regroupement en une seule opération empêche le compilateur de pouvoir gérer indépendamment les calculs et les communications dans les étapes d'optimisation ultérieures (vectorisation et anticipation des communications).

5.2.3. *Optimisation du schéma de compilation*

Motivation Le schéma de compilation présenté précédemment est basé sur la transformation de chaque affectation du programme séquentiel. Cette granularité est source d'inefficacité dès lors que l'on compile des boucles comme par exemple

$$\begin{array}{l} \textbf{pour } i = 1\ ,\ 1000 \\ \quad \textbf{pour } j = i\ ,\ 2{*}i{+}1 \\ \quad\quad S(i,j)\ :\ A[\,i\ ,\ j{-}i\,] := B[\,j\ ,\ 2{*}i{-}2\,] \end{array}$$

puisqu'il n'y a pas, à la compilation, d'analyse des domaines de données accédées. En effet, chaque processus doit effectuer des tests, et ce pour chaque vecteur d'itération (i, j) de la boucle, pour savoir s'il doit émettre ou recevoir $B[\,j\ ,\ 2{*}i{-}2\,]$ et s'il doit réaliser l'affectation $S(i, j)$.

Le schéma est particulièrement inadapté au contexte des boucles parallèles dans lequel toutes les lectures peuvent se faire avant les écritures et donc dans lequel on peut réaliser la séparation des codes d'échange de données et de calcul. C'est le cas de la boucle précédente puisque les $S(i, j)$ peuvent s'exécuter dans n'importe quel ordre.

Principe Un nouveau schéma de compilation réalisant la séparation des codes d'échange de données et de calcul a donc été défini pour les boucles parallèles. Il incorpore une analyse de domaines qui exploite le partitionnement par blocs des tableaux et qui permet la factorisation des tests sur :

. des séquences d'émissions puis de réceptions d'éléments de tableau entre deux processus, ce qui entraîne une vectorisation naturelle des communications entre les processus.
. des séquences d'affectations locales à un processus.

À noter que cette analyse est symbolique, ce qui signifie que la complexité de la production de code est indépendante du nombre de processeurs. Elle repose par ailleurs sur des techniques qui sont également utilisées pour optimiser l'exécution de programmes parallèles sur machines à mémoire partagée [FEA 92, DAR 93].

Précisons tout cela sur l'exemple précédent, avec les tableaux :

. $A[0..3999][0..3999]$ partitionné en 8 blocs 500×4000 numérotés de 0 à 7
. $B[0..3999][0..3999]$ partitionné en 8 blocs 4000×500 numérotés de 0 à 7

Synthèse du code d'échange de données

1. Pour chaque bloc k_A de A et pour chaque bloc k_B de B ($k_A, k_B \in 0..7$), on va s'intéresser au calcul du sous-domaine d'itération réalisant les écritures sur k_A et les lectures sur k_B. On peut montrer que l'ensemble de ces sous-domaines peut être caractérisé par un unique polyèdre $\mathcal{E}$, défini comme étant l'ensemble des quadruplets $< k_A, k_B, i, j >$ tels que $S(i,j)$ provoque une écriture sur le bloc k_A et une lecture sur le bloc k_B.
2. À tout polyèdre (borné non vide), on peut associer un nid de boucles produisant l'énumération de ses points. Pour $\mathcal{E}$, on obtient

   pour $k_A = 0, 2$
   pour $k_B =$ **max**$(0,$ **div**$(500*k_A-1,250)),$ **min**$(3,$ **div**$(250*k_A+249,125))$
   pour $i =$ **max**$(250*k_B+1, 500*k_A),$ **min**(**div**$(500*k_B+501,2), 500*k_A+499)$
   pour $j = i, 2*i+1$

   Les deux premières boucles du code d'énumération généré pour l'exemple nous montrent que tous les couples de blocs (k_A, k_B) de $0..7 \times 0..7$ ne sont pas atteints par le calcul.
3. En insérant des gardes dans le code d'énumération, on génère les codes SPMD d'émission puis de réception des éléments de B lus dans le calcul. Pour l'émission, on produit ainsi

   pour $k_A = 0, 2$
   si `myself` $\neq$ possesseur du bloc k_A de A **alors**
   pour $k_B =$ **max**$(0,$ **div**$(500*k_A-1,250)),$ **min**$(3,$ **div**$(250*k_A+249,125))$
   si `myself` $=$ possesseur du bloc k_B de B **alors**
   pour $i =$ **max**$(250*k_B+1, 500*k_A),$ **min**(**div**$(500*k_B+501,2), 500*k_A+499)$
   pour $j = i, 2*i+1$
   envoyer $B[j, 2*i-2]$ au possesseur du bloc k_A de A

   Remarque On a ici décrit les émissions élément par élément ; en pratique, ces émissions sont vectorisées.

Synthèse du code de calcul

1. Pour chaque bloc k_A de A ($k_A \in 0..7$), on va s'intéresser dans cette phase au calcul du sous-domaine d'itération réalisant les écritures sur k_A. On peut, comme précédemment, montrer que l'ensemble des sous-domaines peut être caractérisé par un seul polyèdre $\mathcal{C}$ dont les points sont les triplets $< k_A, i, j >$ tels que $S(i,j)$ provoque une écriture sur le bloc k_A.

2. On synthétise alors le code d'énumération des points de $\mathcal{C}$:

```
pour k_A = 0 , 2
  pour i = max(500*k_A,1) , min(500*k_A+499,1000)
    pour j = i , 2*i+1
```

qui montre que les blocs de A numérotés de 3 à 7 ne sont pas atteints par le calcul.

3. On en déduit, par insertion d'une garde, le code SPMD de calcul

```
pour k_A = 0 , 2
  si myself = possesseur du bloc k_A de A alors
    pour i = max(500*k_A,1) , min(500*k_A+499,1000)
      pour j = i , 2*i+1
        A[ i , j-i ] := B[ j , 2*i-2 ]
```

Évaluation Les premiers résultats font apparaître l'efficacité du code d'échange de données généré. Par contre, l'efficacité du code de calcul est étroitement liée à la gestion mémoire utilisée à l'exécution.

5.2.4. *Gestion mémoire*

Les choix de mise en œuvre de la représentation des blocs locaux des tableaux distribués, des accès aux éléments de ces blocs ainsi que la gestion des zones mémoires destinées à recevoir des éléments distants ont une influence non négligeable sur l'efficacité du code produit.

Accès aux données locales Les opérations Refresh et Exec nécessitent le calcul du possesseur d'un élément de tableau distribué et, pour le possesseur, l'accès à l'emplacement local correspondant à cet élément. La représentation locale à un processus d'un tableau distribué à n dimensions est un vecteur composé de l'ensemble des blocs du tableau initial associés au processus par la distribution. Le compilateur génère une *table des blocs* où sont stockés, pour chaque bloc, le numéro du possesseur et son adresse locale. Pour un accès, le compilateur transforme les indices initiaux $(i_0, \ldots, i_{n-1})$ en un couple (B, I) d'expressions portant sur ces indices. B représente le numéro du bloc contenant l'élément et I l'indice de l'élément dans le bloc linéarisé. À l'exécution, l'évaluation du couple (B, I) et un passage par la table des blocs permettent de déterminer le possesseur de l'élément et éventuellement d'accéder à l'emplacement mémoire correspondant.

Le surcoût mémoire de la représentation choisie est faible. Cependant, pour certains partitionnements, le coût d'un accès, dominé par le calcul de B et de I, peut s'avérer important.

Mémoire de communication Dans le schéma de compilation de base, lors de la traduction d'une affectation, une variable temporaire est utilisée pour stocker la valeur de chaque référence en lecture à une variable distribuée. La durée de vie de ces temporaires est limitée à l'exécution de l'affectation. Le nombre de temporaires nécessaires

reste donc faible, on peut les allouer un à un et y accéder directement par leur nom. Ainsi l'affectation

$$x = B[j] + C[j+1]$$

dans laquelle le scalaire x est dupliqué et les tableaux B et C sont partitionnés en blocs de 10 éléments sera compilé en

```
{
 int tmp1, tmp2;

 refresh_rep(tmp1, B, j div 10, j mod 10);
 refresh_rep(tmp2, C, (j+1) div 10, (j+1) mod 10);

 exec_rep(x, tmp1 + tmp2);
}
```

Les expressions sur j correspondent aux couples (B, I). Les temporaires sont alloués par le mécanisme d'ouverture de bloc (`{`) et la déclaration de deux variables locales à ce bloc ; `refresh_rep` et `exec_rep` sont des primitives de l'exécutif.

Prise en compte du schéma optimisé L'application du schéma de compilation optimisé défini précédemment pose de nouveaux problèmes : les communications sont groupées et dans la phase de calcul, il n'y pas de distinction a priori entre les lectures d'éléments locaux et les lectures d'éléments reçus. De plus, les communications devenant plus efficaces, le coût de l'accès par évaluation du couple (B, I) a une importance relative accrue. La mise au point d'un nouveau mécanisme d'allocation et d'accès aux données à la fois locales et reçues est actuellement en cours pour résoudre ces problèmes.

5.3. *L'exécutif Pandore II*

L'exécutif Pandore II permet l'exécution du code produit par le compilateur sur différentes APMD. Il met en œuvre le modèle de machine d'exécution adopté (gestion des processus, communication par canaux FIFO, réseau complètement maillé) à l'aide des primitives offertes par le système d'exploitation de l'architecture cible. Il assure la gestion mémoire : il parachève notamment la mise en œuvre des accès aux éléments des tableaux distribués.

L'exécutif est constitué d'un ensemble de primitives écrites sous la forme de macro-instructions `cpp` ; il est organisé selon deux niveaux : le premier assure le lien avec le compilateur (le code généré ne fait appel qu'aux primitives de ce niveau). Le second niveau forme l'interface avec l'APMD cible.

La séparation compilateur/exécutif offre une plus grande souplesse d'évolution de l'environnement de programmation Pandore II. Elle permet d'expérimenter rapidement des optimisations du schéma de compilation ou des optimisations de plus bas niveau liées à l'accès aux données. De plus, la structuration de l'exécutif facilite le portage vers d'autres APMD : seule la couche logicielle de plus bas niveau doit être réécrite. Ainsi des versions de Pandore II pour plusieurs APMD ont déjà été écrites.

6. Comparaison des trois approches

6.1. *Au niveau du langage*

6.1.1. *Le pouvoir d'expression de la distribution*

Bien que les fonctions de décomposition offertes par les trois langages soient similaires (par blocs ou cycliques), les langages Vienna Fortran et Fortran D ont un pouvoir d'expression de distribution supérieur à celui de C-Pandore II. Ils possèdent notamment la notion d'*alignement* qui permet de caractériser la distribution d'un tableau en fonction de celles d'autres tableaux et offrent également la possibilité de dupliquer des blocs selon certaines dimensions.

6.1.2. *La boucle* `FORALL`

La boucle `FORALL` de Fortran D est une généralisation des opérations vectorielles de Fortran 90. L'intérêt de cette construction dans le cadre de la distribution de programmes séquentiels ne semble pas évident car la plupart des optimisations réalisées par les compilateurs se basent sur le parallélisme intrinsèque des boucles, que ce soient des boucles `DO` ou des boucles `FORALL`.

La boucle `FORALL` de Vienna Fortran est une véritable boucle parallèle. Elle permet d'indiquer au compilateur une absence de dépendance qui pourrait ne pas être détectée automatiquement.

Dans les deux cas, la possibilité d'exprimer des opérations de réduction a été ajoutée. Ceci permet aux compilateurs d'optimiser le traitement de telles opérations.

6.1.3. *Les procédures*

Les procédures sont traitées avec plus ou moins de souplesse dans les trois approches présentées. Le langage Vienna Fortran est le moins restrictif : il autorise l'imbrication d'appels (non récursifs), les paramètres pouvant hériter de la distribution du contexte d'appel ou être redistribués à l'entrée de la procédure. Dans Fortran D et C-Pandore II, les paramètres des procédures sont redistribués à l'entrée et recouvrent en sortie, leur distribution d'avant l'appel. Contrairement à C-Pandore II, Fortran D permet l'imbrication d'appels.

6.2. *Au niveau de la compilation*

6.2.1. *Sous-ensemble compilé*

La richesse des langages Vienna Fortran et Fortran D a contraint les écrivains des compilateurs à restreindre les langages compilés.

Les limitations les plus sévères sont celles adoptées par le compilateur Fortran D actuel : les distributions acceptées ne décomposent qu'une seule dimension du tableau (`BLOCK` ou `CYCLIC`) : la duplication selon une dimension donnée n'est pas traitée ; la taille des tableaux, les valeurs des bornes de boucles, le nombre de processeurs

sont des constantes connues à la compilation ; les expressions d'indices des références tableaux doivent toutes être de la forme $i + cte$ où i est une variable d'itération et cte une constante entière dont la valeur est connue à la compilation ; les pas d'itération doivent tous être égaux à 1.

Le compilateur VFCS quant à lui, se limite au traitement des distributions statiques en blocs : les distributions cycliques ne sont pas compilées. Les paramètres des procédures ne peuvent qu'hériter leur distribution : toute distribution explicite des paramètres formels est interdite, ce qui limite à un seul niveau l'analyse de distribution inter-procédurale.

Le langage C-Pandore II, plus limité, est compilé entièrement. Ainsi, les distributions en blocs ou cycliques d'une ou plusieurs dimensions sont traitées.

6.2.2. *Sous-ensemble optimisé*

Les trois compilateurs optimisent des boucles parallèles dont la forme est plus ou moins contrainte. Dans Fortran D, les bornes des boucles sont des constantes et les indices des références aux tableaux sont de la forme $i + cte$. Dans Vienna Fortran et Pandore, les bornes des boucles et les fonctions d'accès aux tableaux sont affines (à une seule variable pour Vienna Fortran). Vienna Fortran et Fortran D imposent que les tableaux manipulés dans les boucles soient partitionnés en blocs placés de telle façon qu'un processeur ne possède qu'un bloc d'un tableau donné. Le système Pandore traite les distributions où un processeur possède plusieurs blocs d'un même tableau, notamment les distributions cycliques.

Afin d'optimiser la compilation des boucles, les trois systèmes se fondent sur une analyse des domaines de données accédés dont la nature et la précision diffèrent selon les approches. L'analyse symbolique réalisée par le compilateur Pandore permet la production d'un véritable code SPMD : le temps de compilation et la taille du code produit sont indépendants du nombre de processeurs exécutant le code. Par contre, les systèmes Vienna Fortran et Fortran D procèdent à une analyse par processeur. De plus, la taille du code produit est fonction du nombre de processeurs. En ce qui concerne la précision de l'analyse, Pandore utilise des polyèdres pour la représentation mathématique des domaines (peu d'approximations sont faites). Les DSR de Fortran D et a fortiori, les descripteurs de recouvrement utilisés par le compilateur VFCS, engendrent des approximations prohibitives dans un cadre général.

Il est à noter que Vienna Fortran est le seul des trois systèmes à intégrer le traitement des boucles parallèles comportant des accès irréguliers grâce au mécanisme de l'inspecteur/exécuteur. On peut cependant s'interroger sur la viabilité de cette technique.

6.3. *Environnement*

Contrairement à Pandore, les systèmes Vienna Fortran et Fortran D intègrent des outils d'analyse et de transformation classiques de programmes : analyse de flot de données, test de dépendances, normalisation d'expressions, transformation de boucles.

De tels outils semblent en effet nécessaires si l'on souhaite compiler des applications quelconques.

Actuellement, seul Pandore dispose d'un véritable support d'exécution pour les programmes compilés. Ce support facilite la portabilité et permet d'optimiser les accès aux données sans avoir à remanier le compilateur. Le code généré par les deux autres compilateurs contient déjà des appels à des primitives de communication : la portabilité de ce code repose donc entièrement sur le choix de ces primitives.

Différents outils annexes ont été développés par les trois projets. Vienna Fortran et Fortran D ont chacun mis en œuvre un estimateur statique de performances [FAH 92, BAL 91]. Pandore incorpore des outils d'analyse d'exécution utilisés pour étudier le comportement des programmes parallèles générés. De plus, des travaux portant sur la validation des techniques de transformation adoptées par le compilateur Pandore ont été réalisés [BAR 93].

7. Conclusion

Actuellement, la programmation des APMD est une tâche complexe, nécessitant de la part de l'utilisateur une bonne connaissance de la machine. Plusieurs pistes peuvent être explorées pour améliorer cette situation. La compilation de programmes séquentiels guidée par la distribution des données semble être une approche convenant bien au calcul scientifique, elle apporte une aide notable au programmeur en le déchargeant du codage des processus parallèles et de leur coopération. Plusieurs prototypes de compilateurs montrent l'intérêt de cette approche mais, à l'heure actuelle, ils ne permettent d'obtenir de bonnes performances que pour une classe réduite d'algorithmes. Étendre leur efficacité exige de développer des techniques d'analyse sophistiquées au niveau de la compilation et de construire des schémas d'exécution adaptés. Quand les prototypes auront atteint un degré de maturité plus grand, il sera nécessaire de passer à l'expérimentation sur des applications réelles afin d'apprécier le pouvoir d'expression des directives de distribution et d'évaluer les performances obtenues.

8. Bibliographie

[AND 90] ANDRÉ F., PAZAT J-L. et THOMAS H., « Pandore: A system to manage data distribution », *International Conference on Supercomputing*, ACM, June 11-15 1990.

[AND 92] ANDRÉ F., CHÉRON O., PAZAT J-L. et THOMAS H., « Efficient code generation for distributed memory machines », *Parallel Computing '91*, Parallel Computing Society, Elsevier Science Publishers B.V., September 1992.

[AND 93] ANDRÉ F., CHÉRON O. et PAZAT J-L., « Compiling Sequential Programs for Distributed Memory Parallel Computers with Pandore II », *Environments and Tools for Parallel Scientific Computing* éd. par Jack J. Dongarra et Bernard Tourancheau , p. 293–308, Elsevier Science Publishers B.V., 1993.

[BAL 89] BALASUNDARAM V., KENNEDY K., KREMER U., MCKINLEY K. et SUBHLOCK J., « The ParaScope Editor : An interactive parallel programming tool », *Supercomputing 89*, November 1989.

[BAL 90] BALASUNDARAM V., « A mechanism for keeping useful internal information in parallel programming: The Data Access Descriptor », *Journal of Parallel and Distributed Computing, special issue on Software Tools for Parallel Programming and Visualization*, June 1990.

[BAL 91] BALASUNDARAM V., FOX G., KENNEDY K. et KREMER U., « A Static Performance Estimator to Guide Data Partitioning Decisions », *The Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1991.

[BAR 93] BAREAU C., CAILLAUD B., JARD C. et THORAVAL R., « Correctness of automated distribution of sequential programs », *PARLE '93*, 14–18 June 1993.

[BER 90] BERRYMAN H., SALTZ J. et SCROGGS J., *Execution time support for adaptive scientific algorithms on distributed memory machines*, Rapport technique n° 90-41, ICASE, May 1990.

[BOM 90] BOMANS L. et HEMPEL R., « The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2 », *Parallel Computing 15*, p. 119–132, North-Holland, 1990.

[BRE 92] BREZANY P., CHAPMAN B. M. et ZIMA H. P., *Automatic Parallelization for GENESIS*, Rapport technique n° ACPC/TR 92-16, Austrian Center for Parallel Computation, November 1992.

[CAL 88] CALLAHAN D. et KENNEDY K., « Compiling programs for distributed-memory multiprocessors », *Journal of Supercomputing*, vol. 2, 1988, p. 151–169.

[CHA 91] CHAPMAN B., MEHROTRA P. et ZIMA H., *Vienna Fortran: A Fortran Language Extension for Distributed Memory Multiprocessors*, Rapport technique n° 91-72, ICASE, September 1991.

[DAR 93] DARTE ALAIN, RISSET TANGUY et ROBERT YVES, « Loop nest scheduling and transformations », *Environments and Tools for Parallel Scientific Computing* éd. par Dongarra Jack J. et Tourancheau Bernard, p. 309–332, Elsevier Science Publishers B.V., 1993.

[DAS 92] DAS R., SALTZ J. et BERRYMAN H., *A Manual for PARTI Runtime Primitives - Revision 1*, Rapport technique, Langley Reseach Center, Hampton VA 23065, ICASE, December 1992.

[DR 85] DAREMA-RODGERS F., NORTON V.A. et PFISTER G.F., *Using a Single-Program-Multiple-Data Computational Model for Parallel Execution of Scientific Applications*, Rapport technique n° RC11552, IBM T.J Watson Research Center, November 1985.

[FAH 92] FAHRINGER T., BLASKO R. et ZIMA H.P., « Automatic Performance Prediction to Support Parallelization of Fortran Programs for Massively Parallel Systems », *Proceedings of the '92 International Conference on Supercomputing*, p. 347–356, ACM press, July 1992.

[FEA 92] FEAUTRIER PAUL, *Towards Automatic Distribution*, Rapport technique n° 92.95, IBP/MASI, December 1992.

[FOR 93] FORUM HIGH PERFORMANCE FORTRAN, *High Performance Fortran Language Specification*, Rapport technique n° Version 1.0, Draft, Rice University, January 1993.

[GER 90] GERNDT M., « Updating distributed variables in local computations », *Concurrency Practice and Experience*, 1990.

[GIL 88] GILOI W.K., « SUPRENUM: A trendsetter in modern supercomputer developement », *Parallel Computing*, vol. , n° 7, 1988, p. 283–296.

[GUP 92] GUPTA M. et BARNERJEE P., « Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers », *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, n° 2, March 1992.

[HIR 91] HIRANANDANI S., KENNEDY K., KOELBEL C., KREMER U. et TSENG C.W., *An Overview of the Fortran D Programming System*, Rapport technique n° TR91121, Rice University, Center for Research on Parallel Computation, March 1991.

[HIR 92] HIRANANDANI S., KENNEDY K. et TSENG C-W., « Compiling Fortran D for MIMD Distributed-Memory Machines », *Communications of the ACM*, vol. 35, n° 8, August 1992.

[KOE 91] KOEBEL C. et MEHROTRA P., « Compiling global name-space parallel loops for distributed execution », *IEEE Transactions on Parallel and Distributed Systems*, p. 440–451, October 1991.

[LAH 92] LAHJOMRI Z. et PRIOL T., « KOAN : a Shared Virtual Memory for the iPSC/2 Hypercube », *Proceedings of the Second Joint International Conference on vector and Parallel Processing*, p. 441–452, September 1992.

[LI 89] LI K. et SCHAEFER R., « A hypercube shared virtual memory system », *Proceedings of the 1989 International Conference on Parallel Processing*, p. 125–131, 1989.

[O'B 92] O'BOYLE M. et HEDAYAT G.A., « A Transformational Approach to Compiling SISAL for Distributed Memory Architectures », *Third Workshop on Compilers for Parallel Computers*, Austrian Center for Parallel Computation, p. 58–69, July 6–9 1992.

[PAA 90] PAALVAST E. M. et VAN GEMUND A. J., « A method for parallel program generation with an application to the BOOSTER language », *International Conference on Supercomputing*, p. 457–469, June 1990.

[QUI 90] QUINN M. J. et HATCHER P. J., « Compiling SIMD Programs for MIMD Architectures », *ACM Sigplan (PPEALS)*, p. 57–65, 1990.

[RAM 91] RAMANUJAM J. et SADAYAPPAN P., « Compile-time techniques for data distribution in distributed memory machines », *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, n° 4, October 1991.

[ROS 90] ROSING M. et WEAVER R. P., « Mapping data to processors in distributed memory computations », *5' International Conference on Distributed Memory Computing*, April 1990.

[THO 92] THOMAS H., SIPS H. et PAALVAST E., « A taxonomy of user-annotated programs for distributed memory computers », *Proceedings of the 1992 International Conference on Parallel Processing*, August 17–21 1992.

[TSE 93] TSENG C-W., *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*, Thèse de PhD, Rice University, January 1993.

[WOL 89] WOLFE M., *Optimizing supercompilers for supercomputers*, MIT press (Research monographs in parallel and distributed computing), 1989.